Ilo Secure Communication

Security as a universal tool

Liliana O'Sullivan

Table of contents

1. Ilo

Ilo is an end-to-end encrypted messaging platform designed with modern technologies in mind. Users messages are encrypted with the content hidden from the server.

Ilo was created as a 4th year Software Development project at the Institute of Technology Carlow.

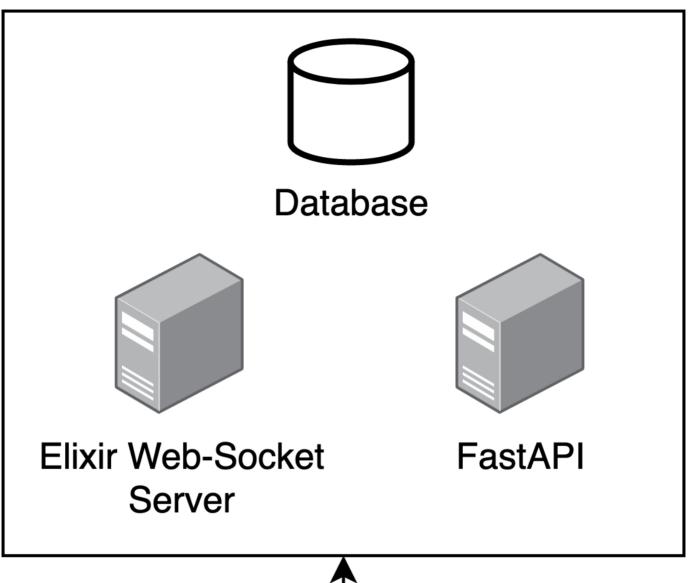**Source Code**: Github

Key Features:

- **Security-First**: Designed with a security-first approach, rather than a second afterthought
- **Standards-based**: Based on (With full compatibility with) open standards for API's, such as OpenAPI/Swagger and JSON Schema
- **Modern Technologies**: Highly performant with Elixir and WebSocket scalability
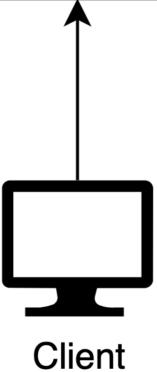
2. Architecture

Ilo is powered predominantly by two of the below technologies.

- **Python** - FastAPI

- **Elixir** - CowBoy

The above technologies are combined in an encapsulated manner, much as showcased below.

Database

Elixir Web-Socket
Server

FastAPI

Client

2.1 Python

Python handles all the 'business' end of the platform. Such as the creation of users or the validation of API keys.

The Python backend is created using FastAPI. FastAPI is a high-performance web framework built on Starlette and Pydantic.

2.2 Elixir

Elixir is referenced as *Potion* within the system. Potion is used as a WebServer for the sending of messages. Elixir is a functional programming language that specialises in concurrency and fault-tolerant systems. The WebSocket server is created using CowBoy.

A room is used to refer to a url identifier after the `/ws/...`. For example, a room for the connection `62.3.11.253/ws/PythonProgramming` would be *PythonProgramming*. Each room has an isolated WebSocket connection, and users connected to a different room do not receive the messages of another room.

2.3 Project layout

The general folder structure can be seen below.

```
.
├── LICENSE
├── docs              # All documentation is stored here
│   ├── docs
│   │   ├── css            # CSS for documentation
│   │   ├── img            # Images used in documentation
│   │   └── js             # JavaScript used in documentation
│   └── mkdocs.yml      # Configuration file for documentation
├── elixir          # Potion system
├── python          # FastAPI
│   ├── Config.py        # Empty class for storing general_config
│   ├── Helper.py        # Helper functions used in this system
│   ├── app.py           # 'main' FastAPI startup file
│   ├── client           # Folder containing legacy reference client
│   ├── models.py        # Contains models used by FastAPI
│   ├── py_client        # Tkinter reference client
│   ├── requirements.txt # Used by pip to install required packages
│   ├── routers          # Contains separated routers used by FastAPI
│   ├── run.sh           # Bash script to run with localhost SSL Certs
```

The above list is not exhaustive and has arbitrary files removed, such as log files for brevity and clarity purposes.

2.4 Additonal Information

2.4.1 Password Hashing

Passwords are hashed using Argon2id by FastAPI and are stored in their hashed state.

3. Configurability

The platform includes two separate configuration files based on YAML

- Logging
- General

3.1 Logging

The logging configuration file contains customisation on how FastAPI will log. This file is created to be in a Python Logging Configuration format. For example, the filename of a log can be changed here, or the logging format can be customised here. A sample of a file can be seen below.

```
formatters:
    simple:
        format: "[%(name)s | %(levelname)s]:%(lineno)s - %(message)s"
    handlers:
        console:
            class: logging.StreamHandler
            formatter: main
            stream: ext://sys.stdout
    loggers:
        user:
            level: DEBUG
            handlers: [console]
```

3.2 General

The general configuration controls variables upon deployment that are desirable to be changed without code changes. For example, Cassandra's address. This file looks as follows.

```
Cassandra_address: "127.0.0.1"
Cassandra_keyspace: "ilo"
Potion_IP: "0.0.0.0:4000"
```

> ✏️ **Note**
>
> The Cassandra keyspace must be created; Ilo will not make it. A keyspace can be created with the following command within the CQL shell
>
> ```
> CREATE keyspace ilo WITH replication={'class':'SimpleStrategy','replication_factor':3};
> ```
>
> The tables used by Ilo will be created automatically.

4. Cryptography

As originally intended, Ilo enables the sending of plaintext and Cryptographic messages. Assuming a valid user exists, is sending the request, Potion will forward the message without verifying keys.

Potion will only enable requests from users that are logged in. A user can log in by sending a put request to `/user` to FastAPI. More information can be found at Swagger. Due to the general-purpose design used within development, Ilo can support any Cryptographic algorithm. The expectation for clients is to use RSA with OAEP (Optimal Asymmetric Encryption Padding).

5. Deployment

Ilo requires FastAPI and Potion running concurrency to function correctly.

FastAPI can be deployed with multiple different WSGI/ASGI web servers. For an ASGI environment, Uvicorn is suggested.

For a WSGI environment, Waitress is recommended. If you are unsure of which to use, ASGI (Uvicorn) is recommended. Additional deployment information can be found here.

FastAPI requires an instance of Cassandra. The Cassandra address and additional information can be configured within the general config

6. Features

Ilo provides the following features.

6.1 Security First

Ilo was designed as a security-first platform, using modern algorithms (Such as Argon2id for hashing) and putting user privacy first. The platform was created as an inspiration for an end-to-end encrypted platform as users increasingly rely on platforms to communicate.

6.2 Standards-based

The entire system is designed around OpenAPI and JSON Schema to conform to common practice in use today. The use of OpenAPI often can allow client code generation in many languages.

6.3 Modern Technologies

All technologies behind the project are based on maximising the capabilities of each technology. Development of the Python codebase utilises type-hinting, new features such as the walrus operator, docstring and more. Elixir's concurrency is utilised to send messages without Python overhead.

7. Swagger & Notes

FastAPI will generate Swagger documentation. This can be accessed with the IP Address and accessing `/docs`, for example, `1.236.66.46:6921/docs`. An example of this can be seen below.

Ilo `In Development` `OAS3`

/openapi.json

A 4th year software development project to create an API that enables secure communication between multiple of its users.

Users All Operations with users profiles.   ⌄

GET	`/user/{username}` Getpublickey
OPTIONS	`/user/{username}` Usernameexists
PUT	`/user` Login a user.
POST	`/user` Creates a user.
DELETE	`/user` Deletes a user.
HEAD	`/user` Deletes a user.

API Keys Manage API Keys. Enables the generation and deletion of keys.   ⌄

| POST | `/key/` Create a API Key |
| DELETE | `/key/{key}` Delete an API Key |

This will showcase the API made available through FastAPI. Additionally, navigating to the `/openapi.json`, for example, `1.236.66.46:6921/openapi.json` will provide a JSON respecting the OpenAPI format of the public-facing functions. An excerpt is shown below

```
{
  "openapi": "3.0.2",
  "info": {
    "title": "Ilo",
    "description": "A 4th year software development project .....",
    "version": "In Development"
  },
  "paths": {
    "/user/{username}": {
      "get": {
        "tags": [
          "Users"
        ],
        "summary": "Getpublickey",
        "operationId": "getPublicKey_user__username__get",
        "parameters": [
          {
            "required": true,
            "schema": {
              "title": "Username",
              "type": "string"
            },
            "name": "username",
            "in": "path"
          }
        ],
```

7.1 JSON Schema

In addition, a JSON Schema is provided for each OpenAPI function. This annotates how a JSON should be formatted to be accepted as a valid request; it can include optional parameters with the request. This information is provided by the Swagger documentation and is provided as part of the OpenAPI JSON. The server will reject the request if the schema is not abided to.

An example of what a Schema may look like is shown below.

User ⌄ {

username*	**string** *title: Username*
password*	**string** *title: Password*
public_key*	**string** *title: Public Key*
api_key*	**string** *title: Api Key*

}

This schema in its' raw format from the OpenAPI JSON is shown below

```
"User":
{
    "title": "User",
    "required": [
        "username",
        "password",
        "public_key",
        "api_key"
    ],
    "type": "object",
    "properties":
    {
        "username": {
        "title": "Username",
        "type": "string"
        },
        "password": {
            "title": "Password",
            "type": "string"
        },
        "public_key": {
            "title": "Public Key",
            "type": "string"
        },
        "api_key": {
            "title": "Api Key",
            "type": "string"
        }
    }
}
```
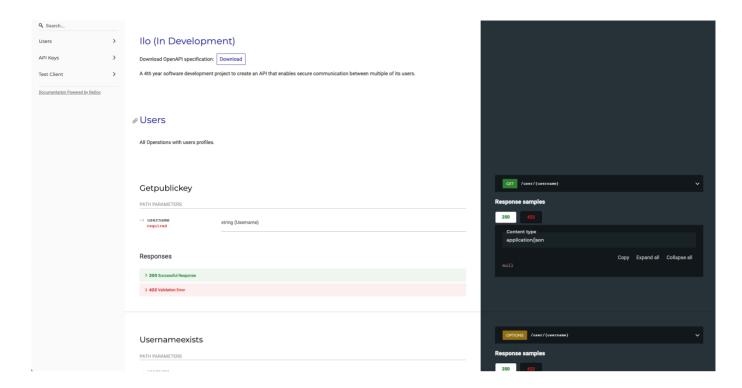
7.2 Noteworthy information

7.3 ReDoc documentation

Documentation in the ReDoc format can be accessed by navigating to `/redoc` .

7.3.1 User Creation

A user must have a password. Ilo's minimum password requirements are

- Length of 8 characters

- Contain minimum 1 number

- Contain minimum 1 lowercase letter

- Contain minimum 1 uppercase letter

- 1 non-alphanumeric character

7.3.2 Unicode support

The platform has support for u[?]$_i c_o d_e$ characters in sending messages and usernames. This allows none-latin characters to be used or support for sending emotes such as 💐😄🦐🍠🥟

8. Development

8.1 Development - Intro

This section will outline development operations on the Ilo platform. This can be in the form of using Ilo as a backend or developing the core platform further.

8.2 API Usage

This section documents the usage of Ilo as a backend; It is highly recommended to have glanced at the architecture section; this section assumes some technical insights of Ilo.

A client must, at minimum, support

- **WebSockets**: This is a requirement for the sending of messages between Potion and the client.
- **JSON Parsing**: The information exchanged within the WebSocket connection is in a JSON format, and as such is essential to the understanding of the information exchanged.

To access public-facing methods, all information is exchanged using an appropriate HTTP method, and a JSON schema. These are documented within the Swagger section.

8.2.1 Message Sending

To send a message, the following pre-conditions must exist.

- An API Key must be obtained 1
- A user must be created 2
- A user must be logged-in 3

Obtaining an API Key

An API Key can be obtained by sending a *POST* request to `/key`. The API will reply with a JSON as follows.

```
{
  "detail": "824a47ae-d0b9-5350-bffb-cc2ee48424a3"
}
```

User Creation

A user can be created by sending a *POST* request to `/user`, with a JSON attached contained the required information. A sample has been provided below. Information on password requirements can be seen here

```
{
  "username": "cookielover57🍪",
  "password": "MySuperSecurePassword57!",
  "public_key": "XzKSzgiX2qoPySbe5T4TSK2018V...",
  "api_key": "824a47ae-d0b9-5350-bffb-cc2ee48424a3"
}
```

Log a user in

To log a user in, we send a PUT request to `/user`. The JSON we send along with this request will look as follows.

```
{
  "username": "cookielover57🍪",
  "password": "MySuperSecurePassword57!",
  "api_key": "824a47ae-d0b9-5350-bffb-cc2ee48424a3"
}
```

8.2.2 Reference Client

A reference client implemented in Python's Tkinter GUI toolkit has been created to provide a sample implementation of a client. This can be found within the Python folder, under `py_client` or the project path of `ilo/python/py_client`.

This client is aimed to showcase the use of the Ilo API. It provided the following functionality.

- Registering a user

- Connecting to a Potion room

- Changing rooms

8.3 Ilo Development

Ilo requires Python and Elixir to function, with both FastAPI and Potion running concurrency to function correctly.

Python can be downloaded from the official Python.org website. Once downloaded and installed, the required Python libraries can be installed from PyPi using pip requirements

```bash
ilo/python $ pip install -r requirements.txt
```

Within the Python folder, there is a bash to run FastAPI with `localhost` certificates. A localhost SSL certificate needs to be created within the python directory. To do this enter the following command at the terminal. Additional information on localhost certificate generation can be found here

```
openssl req -x509 -out localhost.crt -keyout localhost.key \
  -newkey rsa:2048 -nodes -sha256 \
  -subj '/CN=localhost' -extensions EXT -config <( \
  printf "[dn]\nCN=localhost\n[req]\ndistinguished_name = dn\n[EXT]\nsubjectAltName=DNS:localhost\nkeyUsage=digitalSignature\nextendedKeyUsage=serverAuth")
```

The FastAPI server will additionally need a Cassandra backend to function. Specifically, it requires the creation of a keyspace as specified in the `general_config.yaml/Cassandra_keyspace`

The Cassandra address and keyspace can be specified in the `general_config.yaml` file within the python folder. All tables will be created automatically by FastAPI.

A bash script is created to start the FastAPI server with the localhost SSL certificate. It will start the server on port 7999.

```bash
ilo/python $ bash run.sh
```

Potion dependencies can be installed using `mix deps.get` from the potion folder

```bash
ilo/elixir $ mix deps.get
```

Potion can be launched in interactive mode with `iex`

```bash
bash


ilo/elixir $ iex -S mix
```

8.3.1 Potion -> FastAPI communication

Potion does communicate to FastAPI. This occurs once upon user connection to validate a user is logged in to enable communication. This is a HTTP call made to FastAPI. This can be found in the `python/routers/Potion.py` . It has Potion's IP Address whitelisted from the `python/general_config.yaml`